Une démarche orientée modèle pour déployer des systèmes logiciels répartis

Jérémy Dubus — Areski Flissi — Nicolas Dolet — Philippe Merle

Equipe GOAL / ADAM
USTL — LIFL UMR CNRS 8022 — INRIA Futurs
40 avenue Halley, Bât. A, 59650 Villeneuve d'Ascq France
Jeremy.Dubus,Areski.Flissi@lifl.fr — Nicolas.Dolet,Philippe.Merle@inria.fr

RÉSUMÉ. *Le déploiement de systèmes distribués met en jeu de nombreuses technologies hétérogènes. L'administrateur système doit 1) maîtriser le déploiement de chaque logiciel 2) l'adapter aux propriétés des machines, et 3) l'exécuter en respectant l'ordre des dépendances. Ces tâches sont fortement propices aux erreurs. Dans cet article, nous présentons DeployWare, une approche à base de modèles pour le déploiement de systèmes distribués complexes. Cette approche repose sur un méta-modèle en deux parties. La première permet de décrire les propriétés, les dépendances, et actions à effectuer pour déployer des logiciels. La seconde permet d'assembler des instances de logiciels. Ces deux parties sont réalisées de manière à rendre possible la validation comportementale des procédures de déploiement et des systèmes. Les modèles DeployWare sont projetés vers une plate-forme d'exécution à base de composants qui gère automatiquement l'hétérogénéité des machines et l'orchestration des dépendances.*

ABSTRACT. *Deployment of distributed systems involves many heterogeneous technologies. The system administrator has to 1) master the deployment of each technology 2) adapt it to machine properties 3) execute it in respect with order dependencies. These tasks are strongly prone to errors. In this article, we present DeployWare, a model-based approach for complex distributed systems deployment. This approach relies on a metamodel split in two parts. The first allows to describe properties, dependencies, and actions to perform to deploy software. The second allows to compose many software instances. This metamodel allows some behavioural deployment validations to be performed. DeployWare models can be projected onto a component-based execution platform which manages automatically machine's heterogeneity and orchestration of dependencies.*

MOTS-CLÉS : *Déploiement, Validation, Méta-modélisation, Ingénierie dirigée par les modèles*

KEYWORDS: *Deployment, Validation, Metamodeling, Model Driven Engineering*

1. Introduction

Parmi les différentes étapes du cycle de vie des applications réparties, le déploiement reste un véritable problème. Les administrateurs système, en charge du déploiement des applications réparties, doivent constamment s'adapter à la diversité du matériel comme des logiciels. Ces administrateurs doivent faire face à une triple hétérogénéité. La première facette de cette hétérogénéité concerne le support matériel sur lequel les applications sont déployées. Cette diversité concerne différentes propriétés des machines (capacité de calcul et/ou mémoire, système d'exploitation, protocole d'accès à distance, protocole de transfert de fichiers). La seconde facette concerne les paradigmes usités pour réaliser les applications. Ces paradigmes changent au fil des innovations technologiques (*e.g.* programmation par objets, par composants (Szyperski, 1997), de services (Bieberstein *et al.,* 2005), par aspects (Kiczales *et al.,* 2005)). Enfin la troisième facette concerne la variété des implantations d'un même paradigme. À titre d'exemple nous pouvons citer un certain nombre de modèles de composants existants : le modèle Enterprise Java Beans (EJB) (Sun Microsystems Inc., 2003), le modèle CORBA Component Model (CCM) (Object Management Group, 2006a), ou encore le modèle Fractal (Bruneton *et al.,* 2006).

L'administrateur système doit : 1) maîtriser chacun des *protocoles de déploiement* spécifiques à une technologie donnée —*i.e.* les sous-tâches de déploiement pour les logiciels de ces technologies, ainsi que la suite d'instructions élémentaires qui les composent, 2) adapter ces procédures aux caractéristiques des machines censées les exécuter, et enfin 3) exécuter ces protocoles dans un ordre bien défini afin de satisfaire toutes les dépendances entre logiciels. En outre, l'administrateur doit également faire face à la constante augmentation du nombre de machines impliquées comme souligné dans (Flissi *et al.,* 2006).

Nous souhaitons rationnaliser le déploiement, afin de pouvoir l'outiller et l'automatiser au maximum dans le cadre d'une approche conforme à l'ingénierie dirigée par les modèles. Pour cela, nous proposons DeployWare, une démarche à base de modèles pour le déploiement de *Systèmes Répartis Complexes* (SRC). Un SRC représente une architecture logicielle fortement répartie, comprenant toutes les entités logicielles impliquées dans l'exécution d'un système. Ainsi, les bibliothèques, les serveurs, les composants applicatifs métiers ainsi que les multiples dépendances entre ces entités constituent l'architecture d'un SRC. L'architecture globale de notre approche apparaît sur la Figure 1. Dans notre approche, nous proposons un méta-modèle générique de déploiement de SRC composé de deux parties. La première partie (*Expert logiciel*) offre des concepts pour décrire le protocole de déploiement de toute technologie logicielle. La seconde partie (*Administrateur système*) permet d'assembler des instances des logiciels pour définir l'architecture du SRC. En définissant une sémantique pour les concepts du méta-modèle DeployWare, des validations comportementales statiques sont rendues possibles sur le déploiement de SRC. DeployWare rend l'exécution des procédures de déploiement possible grâce à une plate-forme d'exécution qui réifie le processus de déploiement au moyen d'une architecture logicielle de composants, et abstrait cette procédure à la fois des contraintes matérielles, des contraintes

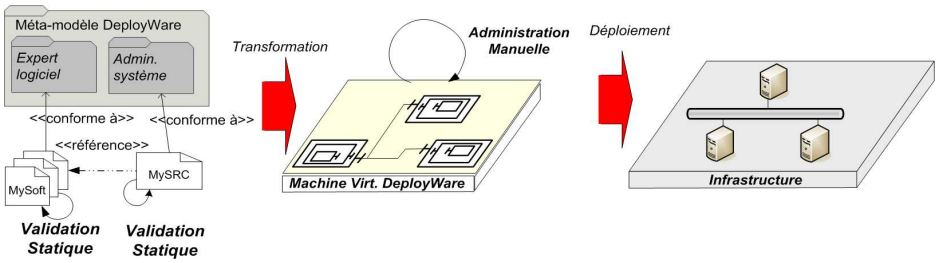

Figure 1. *Vue d'ensemble de notre proposition : DeployWare*

de répartition, mais également des contraintes d'orchestration des tâches du déploiement du SRC.

Ce papier est structuré de la façon suivante. Dans la section 2 nous exposons plus en détails les motivations de ce travail. Dans la section 3, nous exposons les concepts du méta-modèle DeployWare et leurs relations ainsi que différentes validations statiques possibles pour les modèles conformes à ce méta-modèle. Dans cette section nous illustrons aussi l'utilisation de l'environnement de modélisation DeployWare sur un exemple. Dans la section 4, nous détaillons la plate-forme d'exécution à base de composants pour exécuter le déploiement du SRC automatiquement. Dans la section 5, nous dressons un tour d'horizon des travaux existants du domaine, et identifions les apports de DeployWare par rapport à ceux-ci. Enfin dans la section 6, nous concluons cet article et exposons les travaux futurs pour DeployWare avant de conclure.

2. Motivations

Nous allons détailler chacune des motivations à l'origine de notre travail. Ces différentes motivations sont illustrées à l'aide d'un exemple concret de déploiement d'un SRC représenté sur la figure 2. Supposons qu'une entreprise souhaite refondre une partie de son système d'information, en utilisant de nouvelles technologies. Pour cela, elle redéveloppe ses composants métiers conformément aux technologies choisies. Il reste donc ensuite à déployer le SRC. Soient trois machines, une qui héberge le serveur web (M1), une dédiée au serveur d'application (M2), et enfin une dédiée au Système de Gestion de Bases de Données (SGBD) (M3). Les technologies utilisées sont les suivantes, le serveur JEE JOnAS[1] pour héberger des composants métiers d'entreprise, un moteur de Servlet/JSP Tomcat[2] pour héberger les composants Web, et un serveur MySQL[3] comme SGBD. Enfin il est raisonnable d'envisager que l'entreprise possède un service web déployé sur le serveur Tomcat situé sur M1, et utilisant la bibliothèque

1. `http://jonas.objectweb.org`
2. `http://tomcat.apache.org`
3. `http://www-fr.mysql.com`

Axis[4]. Ce service web peut être utilisé par les composants d'entreprise en interne, et peut également rendre les services du SI intégrables directement par le SI d'un partenaire (même service, mais présentation différente par exemple). Les logiciels à déployer sur les machines (relation d'*hébergement*) pour ce SRC sont représentés sur la figure 2, ainsi que les dépendances entre ces logiciels. Ces dépendances sont de deux types. Les dépendances *techniques* représentent des dépendances de fonctionnement, c.-à-d. le lien entre les différentes couches. Elles peuvent représenter les relations entre composants métiers et serveurs d'applications (`war1` et `Tomcat1`), ou entre serveurs d'application et bibliothèques (`jonas1` et `Java-m2`)par exemple. Les dépendances *métier* représentent des dépendances exigées par l'architecture métier du SRC (`war1` et `EJB1`).

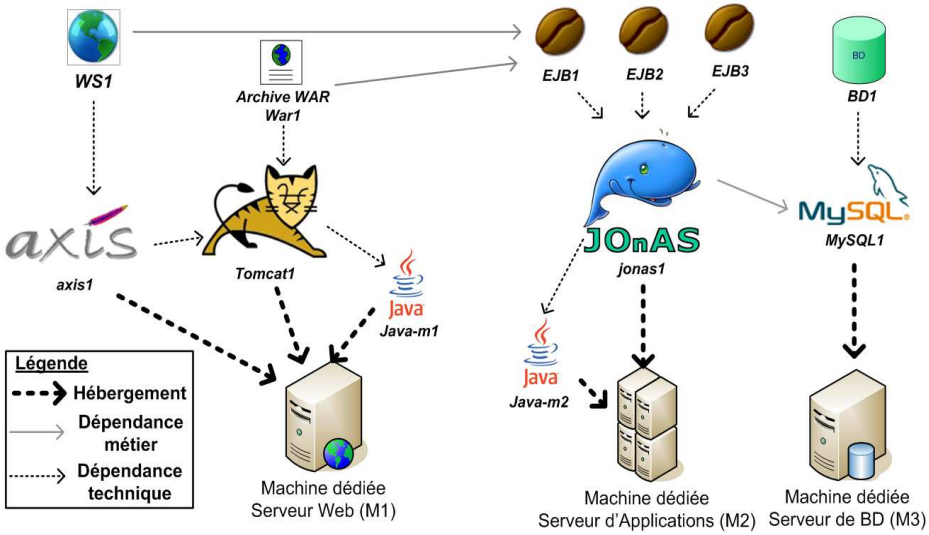

Figure 2. *Un exemple de système distribué complexe d'entreprise*

Nos motivations pour ce travail sont les suivantes. La première motivation consiste à rationnaliser les concepts du déploiement logiciel. Pour ce faire, nous factorisons des concepts de déploiement génériques dans un méta-modèle. La deuxième motivation consiste à séparer les préoccupations. Pour cela, nous découpons notre approche en différents rôles que sont l'**expert logiciel**, l'**administrateur système** et l'**expert réseau**. Notre troisième motivation consiste à apporter un certain nombre de validations aux modèles DeployWare. Ces modèles sont ensuite projetés vers une plate-forme d'exécution en charge du déploiement. Notre quatrième motivation consiste à masquer l'hétérogénéité des machines sur lesquelles le déploiement est accompli. Enfin notre cinquième motivation consiste à automatiser l'orchestration des tâches de déploiement réalisées par la plate-forme.

4. `http ://ws.apache.org/axis`

2.1. *Rationnalisation des concepts de déploiement*

Chaque technologie possède ses propres concepts, ses propres serveurs ou démons à lancer, et de ce fait son propre protocole de déploiement. Cependant, indépendamment de la réalisation concrète (*i.e.* les commandes à exécuter), les actions à entreprendre, quelle que soit la technologie employée, décrivent le même comportement. En effet, pour un logiciel, l'administrateur va d'abord déployer les dépendances en amont de ce logiciel. Il doit entre autres se connecter à distance sur les machines, utiliser le protocole de transfert de la machine pour télécharger les fichiers du logiciel (*e.g.* archives, exécutables), fixer des variables d'environnement, éditer des fichiers de configuration, et enfin lancer des exécutables. Ce protocole générique concerne aussi bien le déploiement d'un serveur JOnAS que celui d'un EJB ou de n'importe quel autre logiciel. La première motivation de ce travail est donc de trouver des concepts génériques qui capturent toutes les opérations à effectuer pour déployer un logiciel, et ce quelle que soit leur réalisation technique. Cette abstraction concerne d'ailleurs toutes les couches logicielles impliquées dans le déploiement d'applications d'entreprises. Ce vocabulaire générique permet de décrire aussi bien le protocole de déploiement des applicatifs métiers, que celui des serveurs d'applications, bibliothèques, voire des systèmes d'exploitation. Ensuite, l'administrateur système décrit l'architecture globale de son système en utilisant ces concepts unifiés pour chaque logiciel qui compose son SRC, quelles que soient les technologies qui sont utilisées.

Disposer d'un vocabulaire commun pour décrire le déploiement de SRC permet d'envisager le déploiement de ces systèmes dans une démarche conforme au paradigme du *Model Driven Architecture* (MDA) de l'OMG (Poole, 2001). En effet, l'utilisation de concepts génériques pour décrire le déploiement d'un SRC correspond à l'utilisation d'un *modèle indépendant des plates-formes* (PIM pour *Platform Independent Model*). Ce PIM est ensuite projeté via une transformation de modèle, vers un *modèle spécifique à une plate-forme* (PSM pour *Platform Specific Model*) prenant en compte les spécificités de la plate-forme sur laquelle on souhaite exécuter le déploiement du SRC correspondant au modèle.

2.2. *Séparation des préoccupations*

Au fil des innovations technologiques tant matérielles que logicielles, la tâche de l'administrateur devient un véritable cauchemar. Mais, de nombreuses tâches ne doivent en fait pas incomber à l'administrateur système seul. Par exemple, la description des différentes étapes de déploiement pour un logiciel donné qui compose son système ne devrait pas être à sa charge, car il n'est pas forcément expert dans l'utilisation de ce logiciel. Dans l'exemple de la figure 2, l'administrateur qui doit déployer ce SRC ne devrait pas apprendre la procédure étape par étape de déploiement du serveur JOnAS `jonas1`, pour accomplir sa tâche. Actuellement une pratique courante consiste à inclure un fichier *ReadMe* dans chaque logiciel. Ce fichier contient les instructions de déploiement du logiciel écrit par les experts du logiciel. L'admi-

nistrateur doit donc exécuter machinalement les instructions du ReadMe en ajustant, si besoin est, certaines propriétés du logiciel. En fait l'administrateur ne devrait avoir qu'à configurer les propriétés d'une instance du logiciel JOnAS et à exécuter automatiquement les opérations de déploiement en fonction de ces propriétés, plutôt que de les réécrire conformément au ReadMe. De la même manière qu'un architecte d'applications à base de composants n'a pas à connaître le contenu des composants qu'il déploie, l'administrateur système qui déploie un SRC n'a pas à connaître le contenu —*i.e.* le protocole de déploiement de chacun des logiciels qu'il déploie. D'autant que cette procédure reste invariablement la même pour un type de logiciel donné ; exiger de l'administrateur qu'il répète ces séquences d'opérations autant de fois qu'il y a d'instances de ce type de logiciel dans son SRC renforce inutilement la complexité de sa tâche. En outre, cette séparation permet une factorisation des descriptions de protocoles de déploiement, ainsi qu'une réutilisation accrue pour le déploiement des logiciels concernés. La deuxième motivation de ce travail qui découle de la première consiste donc à regrouper les concepts génériques de déploiement en différentes catégories qui correspondent aux trois rôles sus-cités (expert logiciel, administrateur système et expert réseau) impliqués dans le déploiement de SRC, et ainsi à définir une séparation claire entre différentes préoccupations relatives au déploiement des SRC.

2.3. *Validation statique du déploiement*

En plus d'avoir à connaître et à écrire les protocoles de déploiement de chaque technologie logicielle, l'administrateur doit actuellement faire face à un certain nombre de précautions à prendre pour chacun des logiciels. Une validation essentielle consiste par exemple à vérifier que chaque sous-tâche de mise en place du logiciel (installation, démarrage, etc.) soit bien associée, au sein d'un protocole de déploiement, à une sous-tâche de suppression du logiciel (désinstallation, arrêt, etc.). Ainsi, s'il existe une tâche d'installation (*resp.* démarrage) il doit exister une tâche de désinstallation (*resp.* arrêt) qui l'annule. Cela assure que tout logiciel déployé est repliable, et que le domaine dans lequel on déploie son application peut redevenir vierge de tout logiciel. Si cette propriété n'est pas respectée, l'allégement de la tâche de déploiement de l'administrateur est annulé par la complexité équivalente de la tâche de repliement. En outre, à l'intérieur même de ces tâches dites inverses, il est possible qu'une tâche d'arrêt (`stop`) d'un logiciel, ne soit pas symétrique en terme d'actions effectuées par rapport à la tâche de démarrage (`start`). Ainsi un effet de bord est introduit à la suite de la séquence démarrer/arrêter, ce qui correspond à un comportement indésirable qui peut d'ailleurs aboutir à des erreurs ou pannes de la machine. Par exemple si dans la tâche `start` on lance un processus, et que dans la tâche `stop` on ne tue pas ce processus, le protocole de déploiement du logiciel est erroné. Il est également important de vérifier qu'un logiciel qui utilise les fonctionnalités d'un autre logiciel dont il dépend (*e.g.* vérifier que le serveur JEE JOnAS, qui utilise les fonctionnalités du Java Runtime Environment (JRE)), est déployé sur une machine où le logiciel JRE est également déployé. À l'inverse un logiciel ne doit pas dépendre d'un autre logiciel duquel il n'utilise aucune fonctionnalité, sous peine de déployer un logiciel inutile sur la ma-

chine. Ces nombreuses vérifications non exhaustives à effectuer représentent autant de lourdes tâches supplémentaires qui incombent à tort à l'administrateur système. Une automatisation de ces vérifications statiques représenterait alors un allégement conséquent du travail de l'administrateur.

2.4. *Masquer l'hétérogénéité des supports d'exécution*

Le protocole de déploiement d'un logiciel dépend grandement des spécificités de la machine sur laquelle il est déployé. Si les intentions restent semblables, la manière de les exprimer est changeante syntaxiquement d'un environnment à un autre. En effet, les commandes à exécuter pour télécharger les fichiers d'un logiciel dépendent du protocole de transfert de fichiers qu'offre la machine (*e.g.* SCP, FTP). De la même manière, les commandes pour fixer des variables dépendent du langage d'accès (le SHELL) à distance fourni par la machine (*e.g.* SH, MS-DOS). Le protocole d'envoi de ces commandes varie également selon les machines (*e.g.* SSH, Telnet). Ainsi l'hétérogénéité d'expression des tâches élémentaires de déploiement par rapport au support matériel qui va l'exécuter est également un obstacle au déploiement automatique de SRC. Masquer cette hétérogénéité dans l'interaction avec le matériel permettrait à l'administrateur de décrire le déploiement de son SRC indépendamment des propriétés et spécificités des machines. L'administrateur déploie une instance du serveur JOnAS `jonas1` sur M2, et les commandes qui doivent être formattées et envoyées à la machine sont inférées à partir des propriétés de la machine M2.

2.5. *Automatiser l'enchainement des dépendances*

La dernière motivation qui nous anime concerne la gestion des dépendances entre logiciels dans le cadre du déploiement. En effet, dans l'exemple de la figure 2, on constate que pour déployer le serveur d'applications JOnAS sur M2 il faut tout d'abord déployer le serveur de bases de données MySQL sur M3. En outre, le serveur JOnAS nécessite également que la machine virtuelle Java soit déployée sur la même machine. L'administrateur doit donc déployer les différents logiciels dans le bon ordre, sous peine de faire échouer la procédure globale de déploiement. Pour un système composé d'une faible quantité d'instances de logiciel cette tâche peut être effectuée à la main. Mais dès lors que le nombre de logiciels augmente, cette tâche requiert énormément de concentration de la part de l'administrateur, voire est impossible à réaliser. La dernière motivation que nous souhaitons traiter, consiste en premier lieu à fournir un moyen simple de décrire les dépendances entre logiciels, aussi bien techniques que métiers. En deuxième lieu, un support d'exécution du déploiement devra pouvoir interpréter ces descriptions pour automatiser la gestion des dépendances, et l'ordre de déploiement des logiciels.

3. Le méta-modèle DeployWare

Dans cette section nous détaillons DeployWare, notre approche à base de modèles pour la description du déploiement de SRC. Dans un premier temps, nous nous penchons sur la définition des concepts de notre méta-modèle, qui doit nous permettre d'abord de décrire le déploiement de n'importe quel logiciel, et ensuite d'assembler des instances de ces logiciels pour former la description du déploiement d'un SRC entier. Nous définissons les concepts généraux de DeployWare ainsi que leurs relations au travers d'un méta-modèle. Nous regroupons ensuite ces concepts selon le rôle de déploiement qu'ils concernent. Dans un second temps, nous nous penchons sur les validations que le méta-modèle nous permet d'apporter. Enfin nous illustrons l'utilisation du méta-modèle à l'aide d'un exemple concret.

3.1. *Des concepts génériques pour le déploiement*

Notre méta-modèle est scindé en deux parties. La première partie offre des concepts utiles à la description du protocole de déploiement pour un logiciel donné. Ces concepts, représentés en haut de la figure 3 (paquetage *TechnoExpert*), sont destinés au rôle d'Expert Logiciel. Afin de définir les concepts généraux de notre méta-modèle, nous avons étudié un multitude de paradigmes et de technologies différents. Parmi les paradigmes étudiés on trouve des technologies dédiées 1) aux architectures orientées services (SOA) comme ActiveBPEL, Orchestra, Apache Tuscany, OW2 PEtALS, 2) à JEE comme les serveurs d'applications OW2 JOnAS, JBoss, Apache Geronimo, Sun GlassFish, 3) aux systèmes à base de CORBA comme OW2 OpenCCM. Nous avons enfin étudié le déploiement de composants métiers pour chacune de ces technologies. La liste exhaustive des technologies est disponible à l'adresse : `http ://fdf.gforge.inria.fr`. De cette étude nous avons créé un ensemble de concepts génériques permettant d'exprimer le déploiement de SRC.

Le concept de **Personality** représente un ensemble de logiciels communs à une technologie donnée. À titre d'exemple, on peut citer la personnalité JOnAS, qui contient non seulement le logiciel serveur JEE, mais également les applications métiers déployées par dessus (*e.g.* archives EAR, WAR, JAR). La notion centrale du méta-modèle est celle de **SoftwareType** qui réifie n'importe quelle entité logicielle mise en œuvre dans un SRC. Un logiciel est hébergé sur une machine dont le type est matérialisé dans le méta-modèle par l'association *hostedBy* avec un **HostType**. Un logiciel peut également dépendre d'autres types de logiciel (relation *dependencies*). Un logiciel comporte des propriétés (**Property**) qui peuvent être intrinsèques à sa définition comme le numéro de port ouvert par un serveur Tomcat (définies par la relation multiple *initial_props*) ou héritées d'un autre logiciel comme la propriété du composant Web qui contient le numéro de port d'un serveur Tomcat dont il dépend (ces propriétés sont définies par l'association *imported_props*. Enfin un logiciel comporte des **Procedures** de déploiement (installation, démarrage, arrêt, désinstallation, etc.) possédant un type et représentant chacune des sous-tâches de déploiement relatives

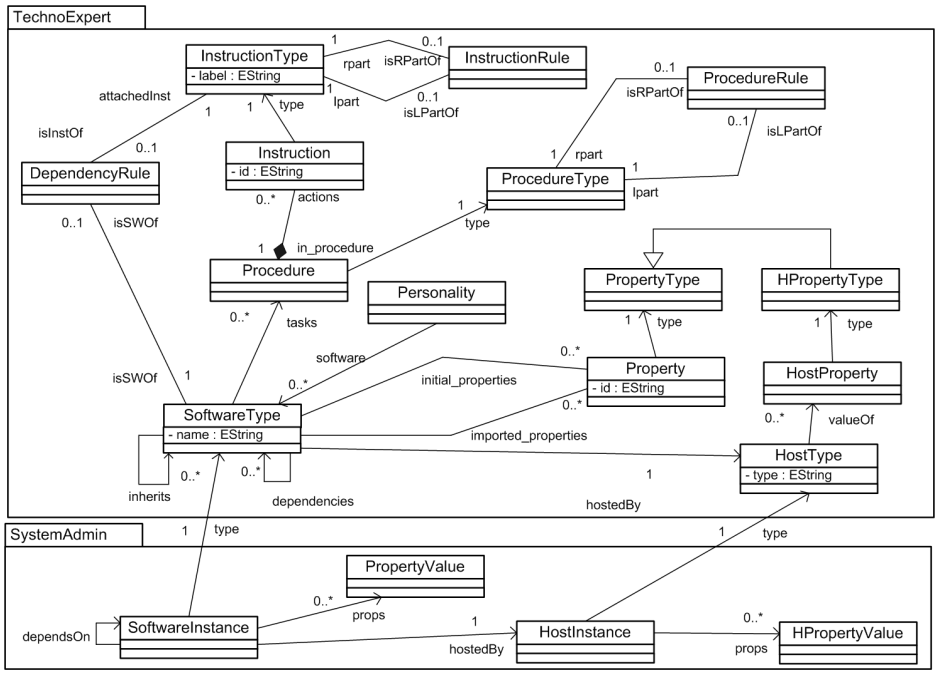

Figure 3. *Vue d'ensemble des deux paquetages du méta-modèle DeployWare*

à ce type de logiciel. Ces procédures sont composées d'**Instructions** (télécharger un fichier, fixer une variable d'environnement, lancer un exécutable, etc.) qui possèdent également un type et représentent des actions élémentaires à effectuer. Un type de logiciel peut également hériter d'autres types de logiciel (relation **inherits**) ce qui signifie qu'il hérite de toutes les propriétés et procédures définies pour ces logiciels.

Une fois les modèles construits par les *experts logiciel*, l'administrateur système construit des instances de logiciels définis dans ces personnalités, configure ces instances et décrit l'assemblage global de ces instances. Un deuxième paquetage dans notre méta-modèle, appelé *SystemAdmin* et représenté dans le bas de la figure 3, offre les concepts permettant à l'administrateur de composer son SRC. Le concept d'instance de logiciel (**SoftwareInstance**) reflète l'instance d'un type de logiciel donné par l'association *type*, cette instance étant hébergée par une instance de machine (**HostInstance**). D'autre part, les propriétés initiales du logiciel doivent être définies (par le biais du concept de **PropertyValue**).

Grâce aux concepts définis dans ces deux paquetages, il est possible 1) de définir les protocoles de déploiement de tout logiciel de manière précise mais indépendamment des contraintes techniques et valider ces modèles 2) d'assembler des instances de ces logiciels, de valider l'assemblage et de les projeter sur une machine donnée afin de produire un plan de déploiement exécutable.

3.2. *Validation statique du déploiement*

Nous allons, dans cette sous-section, mettre en exergue les différentes contraintes à prendre en compte lors de la description des protocoles de déploiement de logiciel. En utilisant les concepts définis dans le méta-modèle présenté précédemment, des erreurs peuvent survenir durant l'exécution du déploiement. Ces erreurs ne sont pas liées à la structure du modèle, mais à la cohérence du protocole de déploiement décrit. Notre proposition consiste à profiter de l'expressivité du méta-modèle pour renforcer la sémantique des concepts par le biais de contraintes sur les modèles DeployWare. Ces contraintes seront alors vérifiables statiquement, et permettront de prévenir l'administrateur d'erreurs potentielles lors du processus global de déploiement.

3.2.1. *Déclaration des dépendances*

Lors de la création de nouveaux types de logiciels, une première difficulté consiste à bien choisir de quels logiciels dépend le logiciel décrit. En effet, oublier une dépendance technique d'un logiciel fera inévitablement échouer le déploiement de chaque instance de ce logiciel. À l'inverse, déclarer une dépendance inutile entrainera, pour chaque instance du logiciel déployé, le déploiement d'un logiciel sans utilité sur la machine. Ainsi il est bénéfique pour l'administrateur qu'une vérification de la pertinence des dépendances soit effectuée. Le méta-modèle permet de définir de nouvelles instructions (*e.g.* la définition d'un type d'instruction `JavaExec` pour le lancement de JVM), et d'associer ces instructions aux logiciels qui fournissent ces instructions sur la machine (par exemple associer le type d'instruction `JavaExec` au type de logiciel `JAVA.JRE`, puisque ce logiciel fournit cette instruction). De cette manière, il est possible de vérifier qu'un logiciel qui utilise dans ses procédures une instruction de type `JavaExec` dépende bien du logiciel `JAVA.JRE` qui fournit cette instruction, sinon il est incorrect. À l'inverse si l'on considère un logiciel qui dépend de `JAVA.JRE`, mais qui n'utilise pas d'instruction de type associé à `JAVA.JRE` dans ses procédures, alors une incohérence est détectée. Néanmoins dans ce cas précis, il est impossible d'affirmer avec exactitude qu'il s'agisse d'une erreur. En effet, on ne peut pas affirmer que cette dépendance est inutile car la dépendance d'un logiciel envers un autre n'implique pas toujours l'utilisation d'une instruction. Par exemple, un serveur JEE JOnAS n'utilise pas directement l'exécutable `java` (*i.e.* `JavaExec`) dans sa procédure de déploiement. Néanmoins sans une JVM installée, JOnAS ne peut pas être déployé. Cette vérification sert donc d'avertissement pour l'expert logiciel, mais ne relève pas une erreur.

3.2.2. *Conformance de types*

Une autre précaution à prendre lors du déploiement de SRC consiste à vérifier que les dépendances d'une instance de logiciel sont bien résolues. En effet, deux types de dépendances existent : les dépendances techniques et les dépendances liées au métier de l'application. Si un logiciel est déployé sans que l'une de ses dépendances techniques ne soit installée, le déploiement échouera immédiatement. En revanche dans le cas d'une dépendance métier non résolue, le logiciel sera correctement déployé, mais l'exécution du métier de l'application sera source d'erreurs. Ainsi lorsqu'un système

est composé de logiciels, il faut vérifier que les dépendances techniques requises par les types de ces logiciels sont bien résolues, ainsi que les dépendances métier. Un tel système décrit au moyen d'un modèle DeployWare peut être validé de la façon suivante : d'une **SoftwareInstance** on peut retrouver le *type* (*i.e.* un **SoftwareType**) et à partir du **SoftwareType** et de l'association *dependsOn*, on peut mémoriser les différents types de Software dont doit dépendre l'instance courante. Dès lors il est possible de vérifier que cette instance de logiciel possède bien dans son association multiple *dependencies* d'autres **SoftwareInstance** avec les types qui conviennent. Cette vérification correspond à une vérification de conformance d'une instance par rapport à son type. On peut également selon le même principe vérifier que les propriétés déclarées pour une instance correspondent bien au type des propriétés du type de logiciel.

3.2.3. *Logiciels non repliables*

Le déploiement est une phase cruciale du cycle de vie d'une application. Un déploiement ne peut être valide que s'il possède la particularité d'être intégralement réversible. Par intégralement réversible, nous entendons que chaque logiciel composant le SRC soit repliable (replier étant l'action symétrique de déployer), et donc le SRC doit pouvoir se replier complètement et laisser le domaine de déploiement dans l'état initial. Un moyen de vérification que chaque logiciel est repliable consiste tout simplement à veiller à ce que les procédures de déploiement d'un logiciel soient bien associées deux à deux : une pour le déploiement avec une pour le repliement. Un exemple simple est le fait de vérifier qu'une procédure de type *démarrage* est bien associée dans le logiciel à une procédure de type *arrêt*. Le méta-modèle DeployWare nous offre les moyens d'opérer de telles vérifications par le biais du concept **ProcedureRule** qui permet d'associer deux **ProcedureType**. Si au niveau *TechnoExpert* on définit qu'une procédure de type *start* est associée à une procédure inverse de type *stop*, on pourra vérifier que tout logiciel comprenant une procédure de type *start* contient bel et bien la procédure inverse de type *stop*.

3.2.4. *Procédures inverses non symétriques*

Pour deux procédures déclarées comme étant de types inverses, il faut vérifier que les actions qu'elles exécutent représentent effectivement des actions qui s'annulent. Toutes les instructions dans une procédure doivent donc trouver leur instruction inverse dans la procédure inverse. De cette manière on peut assurer que les procédures de déploiement écrites sont symétriques. Grâce au méta-modèle DeployWare il est possible d'effectuer cette vérification statiquement sur le même principe que dans les procédures dans la section précédente. L'expert logiciel qui définit des instructions attachées à un logiciel peut associer à l'aide d'une **InstructionRule** deux **InstructionTypes** comme étant des types d'instruction inverses, et pour chaque couple de procédures inverses d'un logiciel, il suffit de vérifier que si l'une de ces instructions est utilisée dans une procédure, l'instruction inverse est présente dans la procédure inverse.

3.2.5. *Conflit de ressources*

Lorsque le nombre de logiciels impliqués est important, il est probable que certains logiciels se retrouvent déployés sur la même machine. Dès lors, afin d'éviter des comportements non désirés de la procédure de déploiement, il faut veiller à ce que tous les logiciels d'une même machine n'entrent pas en conflit de ressources. Les ressources en question peuvent aussi bien concerner le système de fichiers (veiller à ce que deux logiciels ne soient pas déployés dans le même répertoire), ou d'autres ressources comme les ports ouverts à la connexion (veiller à ce que deux serveurs utilisent bien des ports différents). Dans notre méta-modèle DeployWare les propriétés possèdent un type. Il est donc possible, pour chaque **HostInstance** de recenser les logiciels hébergés par la machine correspondante, puis pour chacun de ces logiciels de vérifier que les valeurs des propriétés de même type sont différentes.

3.3. *Illustration*

Dans cette partie, nous illustrons l'utilisation de notre méta-modèle pour modéliser deux types de logiciels impliqués dans le scénario-exemple de la section 2. Nous étudions ensuite cette modélisation afin de juger de la pertinence du méta-modèle. Nous choisissons d'étudier la modélisation type du logiciel JOnAS, ainsi que d'un composant à déployer pour ce type de serveur, *i.e.* un Enterprise Java Bean (EJB). La modélisation (sous forme d'une diagramme d'instances informel) de ces deux types de logiciels est représentée respectivement sur les figure 4 et 5.

Nous considérons dans cet exemple que nous disposons d'un certain nombre d'éléments de modèles génériques insérés dans la bibliothèque de base de DeployWare. Parmi ces éléments, un certain nombre de types d'instructions génériques telles que `SetVariable` pour fixer une variable d'environnement, `UploadGeneratedFile` qui est une instruction réifiant l'action de configurer un fichier de configuration, puis de le télécharger à un endroit donné de l'arborescence de fichiers, et enfin `AddToPath` qui ajoute un chemin à la variable d'environnement `$PATH`. Nous supposons disposer également d'un certain nombre de **ProcedureType** standards, tels que `Installation` et `Uninstallation` qui sont associés via une **ProcedureRule** qui les déclare comme opposés, `Start` et `Stop` déclarés opposés également, et enfin `Configuration`. Finalement, nous supposons disposer d'un type générique *abstrait* de logiciel dit `Installable`, qui factorise des opérations communes à un grand nombre de logiciel, tel que le téléchargement de l'archive, la décompression de l'archive dans un répertoire dans la procédure d'installation, ainsi que les opérations opposées dans la désinstallation (afin de ne pas corrompre la validité du logiciel).

Plaçons nous en premier lieu dans le rôle de l'expert logiciel qui a pour tâche de définir le processus de déploiement du serveur JEE JOnAS. Il définit donc en premier lieu une nouvelle personnalité `JOnAS` dans laquelle il introduit la définition d'un nouveau SoftwareType `SERVER`. Il définit d'abord les propriétés intrinsèques du serveur JOnAS à savoir le port (propriété **http-port**) sur lequel il propose entre autres un ac-

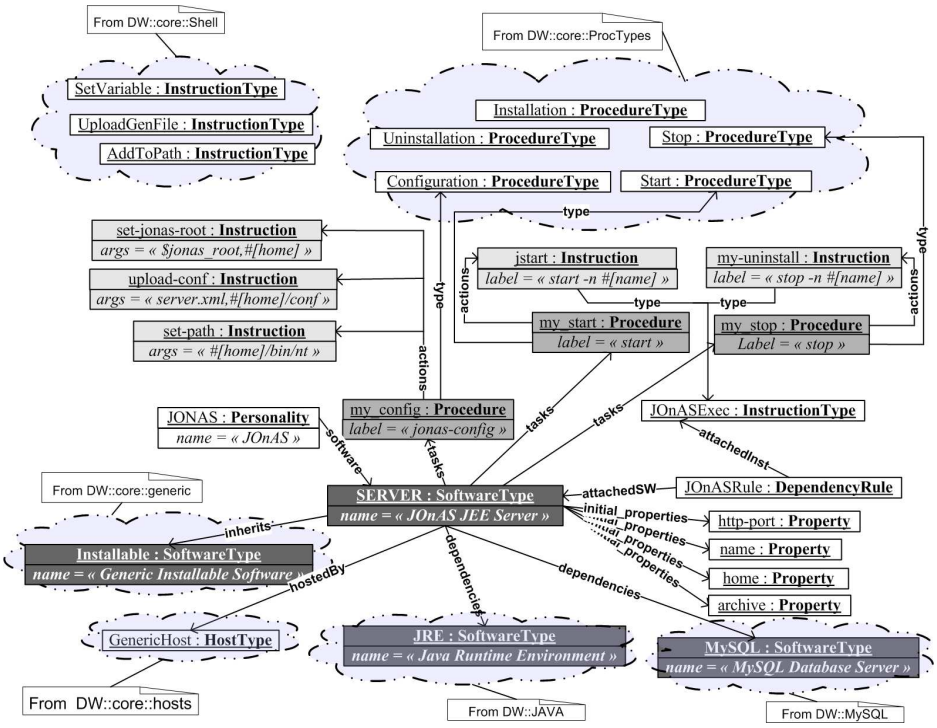

Figure 4. *Modélisation de logiciel avec DeployWare : le serveur JEE JOnAS*

cès d'administration Web, un identifiant (propriété **name**), un répertoire d'installation (propriété **home**) ainsi qu'une archive contenant la distribution (propriété **archive**). Son type de logiciel hérite de **Installable** afin d'hériter des tâches d'installation et désinstallation génériques qui visent à décompresser l'archive à un endroit donné. Il définit ensuite trois procédures : `my_config` de type `Configuration`, `my_start` de type `Start` et `my_stop` de type `Stop`. La procédure de configuration contient trois instructions : pour fixer la variable `$JONAS_ROOT` nécéssaire au démarrage de JOnAS, pour charger un fichier de configuration rempli avec les propriétés du logiciel, et enfin pour ajouter l'exécutable de JOnAS dans la variable d'environnement $PATH. Il définit ensuite un nouveau type d'instruction `JOnASExec` qui réifie l'utilisation de la commande `$JONAS_ROOT/bin/nt/jonas`. Ce nouveau type d'instruction est utilisé dans les procédures `my_start` et `my_stop` afin de respectivement exécuter les commandes pour démarrer (commande `jonas start`) et arrêter (commande `jonas stop`) le serveur.

Le premier point de discussion concerne la procédure de configuration. Nous n'avons pas défini de procédure inverse pour la procédure de configuration, car au sens strict du terme, il s'agit juste d'adapter un logiciel installé, et que théoriquement la désinstallation annihile la configuration en même temps que l'installation. Cepen-

dant dans le cas présent, les opérations de configuration (notamment celles de modification de variables d'environnement) ne sont pas défaites ni dans la désinstallation ni ailleurs. Le logiciel tel qu'il est décrit n'est pas valide, puisqu'un effet de bord apparaît après une séquence install/start/configure/stop/uninstall.

Le deuxième point de discussion concerne la dépendance entre le serveur et le type de logiciel `JRE`. En effet cette dépendance est incluse dans le modèle. Pourtant elle n'est légitimée par aucune des procédures déclarées. En effet dans la personnalité `JAVA` est défini un type d'instruction `JavaExec` réifiant la commande de lancement d'une JVM. Ce type d'instruction n'est utilisé dans aucune procédure du serveur JOnAS. Conformément à la validation des dépendances, ce type de logiciel n'est pas valide puisque la dépendance vers le JRE parait inutile. Néanmoins cette dépendance est bel et bien utile puisque le script `jonas` masque en fait des appels vers un JRE. Nous en déduisons qu'il faut distinguer une dépendance directe d'une dépendance indirecte. Concrètement, un moyen de gérer ce genre de cas consiste à avertir (plutôt qu'interdire) l'expert logiciel que cette dépendance paraît non justifiée. Ainsi l'expert logiciel peut réflechir à la pertinence de cette dépendance et décider ou non de l'enlever. La même remarque est d'ailleurs applicable à la dépendance vers MySQL.

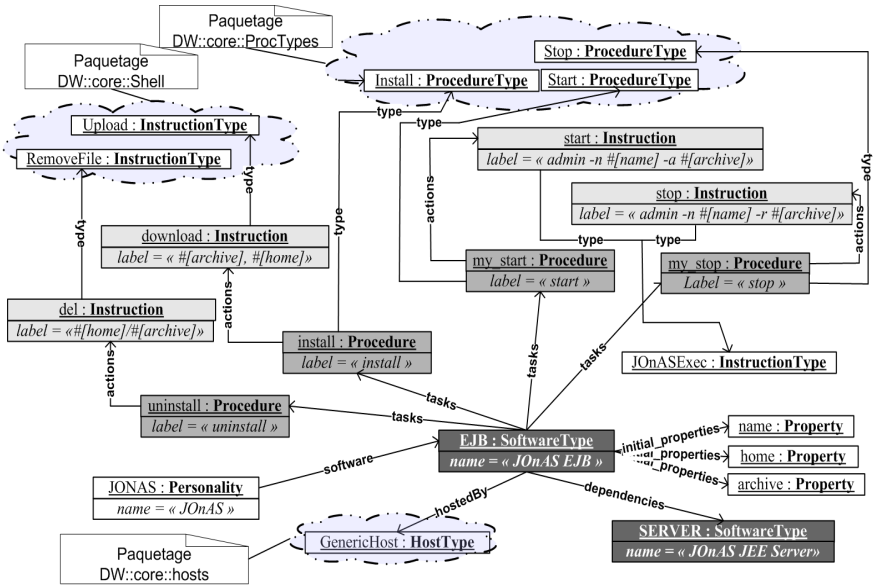

Figure 5. *Modélisation de logiciel avec DeployWare : un EJB*

L'expert logiciel va ensuite définir le modèle de déploiement d'un EJB. Il définit de la même manière les propriétés. Ce logiciel n'hérite pas du logiciel Installable car il n'est entre autres pas nécéssaire de décompresser l'archive d'un EJB. Ainsi une procédure d'installation contient une instruction de téléchargement de l'archive, une procédure de démarrage utilise l'instruction JOnASExec pour déployer l'archive sur le

serveur correspondant à cette instruction, une procédure d'arrêt replie l'archive EJB, et enfin une procédure de désinstallation supprime l'archive. Notons que la dépendance vers le serveur JOnAS est justifiée par la présence de l'instruction JOnASExec.

Il reste à l'administrateur système à assembler les instances de logiciels ainsi définie. Le modèle réifiant une partie de l'exemple de la section 2 est représenté sur la figure 6. La simplicité de ce modèle fait qu'il est très proche de la notation graphique informelle utilisée en figure 2 de la section 2.

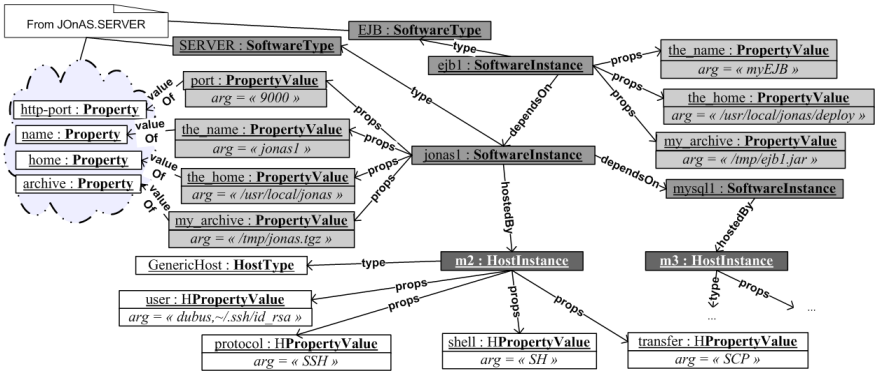

Figure 6. *Modélisation d'un ensemble du SI avec DeployWare*

Ce méta-modèle DeployWare a été créé en utilisant le langage de méta-modélisation Kermeta (Muller *et al.,* 2005). Les validations sont implémentées sous la forme d'un programme Kermeta. Un prototype de plugin pour l'outil de développement Eclipse a été réalisé à l'aide de l'environnement *Eclipse Modeling Framework* (Budinsky *et al.,* 2003). Cet outil permet de créer simplement des instances de modèles DeployWare sous forme arborescente. Ce plugin Eclipse ainsi que le programme permettant de valider les instances DeployWare sont disponibles à l'adresse `http://www.lifl.fr/~dubus/DeployWare` Les personnalités créées sont ensuite projetées vers leur équivalent en composants DeployWare —*i.e* chaque logiciel est transformé selon sa description en composant de logiciel. Le modèle du SRC de la figure 6 est lui projeté en différentes instances des composants logiciels générés configurés par l'administrateur. Le composite global n'a plus qu'à être exécuté pour déployer automatiquement le SI.

4. La machine virtuelle de DeployWare

L'expert logiciel réalise des modèles de logiciels à l'aide du méta-modèle DeployWare. L'administrateur système réalise, lui, des modèles d'assemblage de ces logiciels. Nous définissons maintenant le troisième rôle de déploiement que nous avions évoqué, qui est celui d'expert réseau. Les modèles doivent être projetés afin d'exécuter le déploiement global, fidèlement aux descriptions données, et en tenant compte

des spécificités des machines. Dans cette section, nous détaillons les spécificités de la machine virtuelle DeployWare, en charge d'exécuter ces déploiements. Cette machine virtuelle est capable d'exécuter les processus de déploiement en s'adaptant aux spécificités des machines décrites par l'expert réseau. La plate-forme permet également d'automatiser la gestion des dépendances.

4.1. *Architecture générale*

Le machine virtuelle DeployWare consiste en une plate-forme d'exécution d'applications à base de composants. Chaque concept du méta-modèle est projeté vers un composant. Ainsi, pour construire le processus de déploiement d'un système distribué, il faut assembler ces composants. Les composants de la machine virtuelle DeployWare peuvent être regroupés en trois catégories explicitées dans cette section. Indépendamment de la technologie, par *composant* nous entendons une unité logicielle indépendante, configurable via l'utilisation d'*attributs*. Un composant posséde des *interfaces* dites *fournies* qui offrent des opérations, ainsi que des interfaces dites *requises*. Les interfaces requises sont connectées à des interfaces fournies de même type afin d'appeler des opérations. Nous considérons qu'un composant peut contenir une implémentation, il est alors dit *primitif*, ou alors être composé d'autres composants, il est alors dit *composite*. Un composant peut être contenu par plusieurs composites, lorsque c'est le cas il est dit *partagé*. Nous le verrons dans la suite, le modèle de composants à utiliser pour cette machine virtuelle DeployWare doit être récursif, et posséder la notion de partage de composants. C'est pour ces raisons, et également parce qu'il s'agit d'un modèle de composants léger en terme de performances, que notre choix s'est porté sur le modèle de composants Fractal (Bruneton *et al.,* 2006) et Julia, son implémentation de référence en Java. En outre un langage de définition d'architecture générique a été réalisé pour construire des architectures de déploiement. Ce langage est un sous-ensemble d'un langage de description d'architecture à composants Fractal existant (Fractal ADL). Les constructions de ce langage sont volontairement très réduites : l'opérateur de composition seul permet de construire des architectures. Ecrire une architecture de composants DeployWare signifie créer des composants typés qui contiennent d'autres composants. Les liaisons entre composants à l'intérieur d'un même composant sont automatiquement calculées en fonction du type de composants. L'expression d'architectures écrites dans ce langage est donc simplifiée. La transformation de modèles DeployWare en architectures de composants DeployWare en est donc également simplifiée.

4.2. *Adaptation aux machines*

Un logiciel était composé de composants de Procédure eux-mêmes composés de composants d'Instruction qui requièrent une interface pour exécuter des commandes. Mais, ces commandes doivent être envoyées en utilisant différents protocoles selon que la machine offre un accès en SSH, en Telnet ou autre, et les commandes ont un

certain format selon que le Shell proposé par la machine soit de type SH, CSH ou le Shell de Microsoft Windows. De la même manière le téléchargement des fichiers sur la machine doit se faire conformément au protocole de transfert de fichiers supporté par la machine (SCP, FTP, etc.). L'utilisation de l'interaction entre composants, indépendamment de leur implémentation, va nous permettre de gérer cette adaptation à l'environnement sous-jacent. En effet, les instructions manifestent toujours les mêmes intentions vis-à-vis de la machine (*i.e.* créer un répertoire, naviguer dans le système de fichier, lancer un exécutable, envoyer un fichier). Ainsi ces opérations génériques vont être regroupées dans des interfaces, et différents composants aux implémentations différentes réalisent cette interface, en fonction du Shell pour d'abord formater la commande, puis en fonction du protocole d'accès à distance pour envoyer la commande formattée. Cette architecture est représentée sur la figure 7.

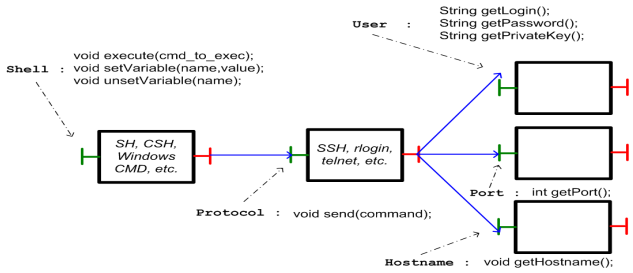

Figure 7. *Les composants d'abstraction des machines de DeployWare*

L'implémentation des composants `Shell` et `Protocol` d'accès aux machines est écrite par le troisième rôle au sein de la séparation des préoccupations du déploiement : l'**expert réseau**. Il est en charge d'implémenter librement chaque opération des interfaces en fonction du mécanisme réseau qu'il réalise (accès, transfert de fichier, etc.). L'administrateur système n'a ensuite plus qu'à décrire les machines du domaine de déploiement concerné en terme d'information de connexion (nom d'utilisateur, mot de passe), le port sur lequel est ouvert un canal de communication, les protocoles d'accès à distance et de transfert de fichier, et le Shell. De son côté l'administrateur système n'a rien d'autre à faire que d'attribuer n'importe quelle instance de logiciel à une machine, la procédure de déploiement de ce logiciel étant définie de manière générique, elle s'exécutera en accord avec les spécificités de la machine. Sur le listing 1 figure l'implémentation de l'interface `Shell` pour le Bourne Shell (SH). Cette classe implémente l'interface `Shell` de la figure 7, en formattant les commandes conformément au langage SH. L'implémentation des interfaces `Protocol` et `Transfer` se fait de manière similaire. L'implémentation consiste à définir respectivement le mode d'envoi des commandes formattées par le composant Shell (*e.g.* via SSH, Telnet), et le mode d'envoi de fichiers binaires (*e.g.* via SCP, FTP). Par exemple, une implémentation existante du protocole SSH repose sur l'API JAVA JSch. Une implémentation qui existe pour l'interface Transfer repose sur l'API Commons Net FTP du consortium Apache.

```
@Component public class SH implements Shell
{
  // L'interface 'protocol' requise par ce composant.
  @Requires protected Protocol protocol;
  public void execute (final String command) {
    // Envoyer la commande via le protocole.
    protocol.send(command);
  }
  public void setVariable (final String name, final String value) {
    // Formatter la commande à la syntaxe shell Unix.
    protocol.send("export " + name + "=" + value);
  }
  public void unsetVariable (final String name) {
    protocol.send("unset " + name);  }  }
```

Listing 1 – L'implémentation SH de l'interface Shell

La projection des **HostInstance** du métamodèle DeployWare revient à définir des `HostInstance`. Ces instances sont configurées à l'aide des composants d'interface `Protocol`, `Shell`, `Transfer` et `User`. Cette dernière interface `User` étant simplement un moyen d'accès au tuple login/mot de passe/clé de sécurité permettant de se connecter à distance sur la machine. Le listing 2 donne un exemple de description possible d'un composant `HostInstance`.

```
host-1 = INTERNET.HOST {
  hostname = INTERNET.HOSTNAME(hostname.mycompany.com);
  user     = INTERNET.USER(login,password,/home/user/.ssh/id_rsa);
  transfer = TRANSFER.SCP;
  protocol = PROTOCOL.OpenSSH;
  shell    = SHELL.SH; }
```

Listing 2 – Description d'une HostInstance

4.3. *Les composants exécutables*

Les composants exécutables implantent les concepts de `Procédure` et d'`Instruction`. Un composant procédure est en charge de l'exécution d'un certain nombre d'instructions dans un ordre donné. Il est donc logiquement composé d'autant de composants d'instruction. Un composant réifiant une instruction représente une action élémentaire de déploiement à effectuer (*e.g.* création de répertoire, lancement d'un script ou téléchargement d'un fichier), il possède une interface requise vers un composant capable d'envoyer la commande à la machine concernée en adaptant cette commande aux propriétés de la machine. Le listing 3 contient l'implémentation du composant de l'instruction pour fixer une variable d'environnement. Ce composant est relié à un composant possédant une interface de type `Shell` (cf. figure 7). Dans le code de ce composant, la méthode setVariable() est appelée sur le Shell auquel il est connecté, indépendamment de l'implantation du composant `Shell`.

```
public class SetVariable extends AbstractRunner
@Component public class SetVariable implements Runnable
{
  // L'interface 'shell' requise par ce composant.
  @Requires protected Shell shell;

  @Attribute protected String name;
  @Attribute protected String value;

  public void run () {
    shell.setVariable(name,value);  } }
```

Listing 3 – L'implémentation de l'instruction SetVariable

4.4. *Les composants de logiciels*

Les composants de logiciel, représentés sur la Figure 8, implantent le concept de SoftwareType. Ils sont composés de différentes informations indispensables telles que les dépendances à déployer en amont, les propriétés du logiciel concerné, et surtout les procédures de ce logiciel. Tout, dans l'architecture de la machine virtuelle DeployWare, est composant. Ainsi les propriétés d'un logiciel sont représentées sous forme de composants possédant une interface fournie générique donnant la possibilité de consulter la valeur de la propriété mais également de la modifier. Les dépendances d'un logiciel sont représentées sous forme d'un composite contenant une référence vers chacune des instances de logiciels dont dépend le logiciel courant (les dépendances sont donc *partagées* à l'intérieur de ce composant). Un composant automate gère le cycle de vie du logiciel et l'exécution du déploiement des dépendances. Le listing 4 présente la description en langage d'architecture DeployWare du type de logiciel JOnAS. Dans cette architecture on définit un type de logiciel appelé `JOnAS.SERVER`. Le sous-composant `properties` regroupe les propriétés à définir pour ce logiciel. Le sous-composant `deployment` regroupe les procédures de ce logiciel.

Le listing 5 présente une instance du composant `JOnAS.SERVER`. Cette instance contient les valeurs pour les propriétés définies dans le type, ainsi que l'information de la machine sur laquelle ce serveur sera déployé, en l'occurrence le **host-1**.

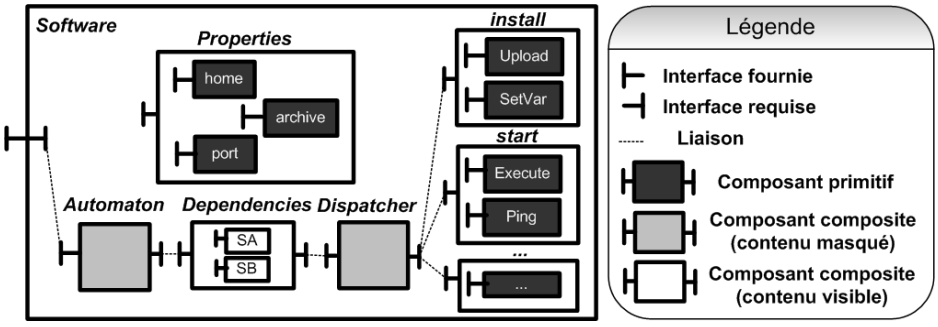

Figure 8. *Le composant Software de la machine virtuelle DeployWare*

```
JOnAS.SERVER = software.Installable, JAVA.DependOn {
  properties {
    archive = JOnAS.ARCHIVE;
    home = JOnAS.HOME;
    http-port = HTTP.PORT(9000);
    name = JOnAS.NAME(JOnAS);     }
  deployment {
    configure {
      TRANSFER.UploadGeneratedFile(server.xml,#[home]/conf);
      SHELL.AddPath(#[home]/bin/nt:#[home]/bin/unix);
      SHELL.SetVariable(JONAS_ROOT,#[home]);       }
```

```
    start {
      JOnAS.JOnASExec(start -n #[name]);     }
    stop {
      JOnAS.JOnASExec(stop -n #[name]);   } } }
```

Listing 4 – Le type de logiciel JOnAS.SERVER

```
jonas-on-host-1 = JOnAS.SERVER {
  properties {
    archive = JOnAS.ARCHIVE (/archives/jonas.zip);
    home = JOnAS.HOME(/usr/jonas);
    name = JOnAS.NAME(jonas1);
    http-port = HTTP.PORT(9000);   }
  host = /host-1; }
```

Listing 5 – Une instance de JOnAS.SERVER

4.5. *Gestion des dépendances*

Les dépendances d'un logiciel sont représentées sous forme d'un composite qui contient des références (via le mécanisme de partage) vers d'autres instances de logiciels du système. Ainsi si un logiciel A qui dépend d'un logiciel B doit être déployé, le fonctionnement du logiciel est le suivant : pour chaque logiciel dans le composite de dépendances la procédure équivalente sera appelée (*i.e.* si *start* est appelée sur A alors *start* sera appelée sur chaque logiciel des dépendances de A). Ainsi le logiciel B sera déployé en amont de A. Récursivement, si A dépend de B, B sera déployé avant A, mais avant d'exécuter la procédure contenue dans B, les dépendances de B seront déployées et ainsi de suite. De la sorte, on assure que chaque logiciel est déployé lui-même après ses dépendances, et si les dépendances sont correctement déclarées par l'expert logiciel, et respectées par l'administrateur système logiciel par logiciel, alors la gestion des dépendances de manière globale est automatique.

4.6. *Implantation*

La plate-forme d'exécution DeployWare à base de composants a été réalisée à partir de composants Fractal en utilisant Julia, l'implémentation de référence en Java du modèle de composants Fractal. Le langage de description d'architecture utilisé repose sur Fractal ADL. La plate-forme d'éxécution DeployWare est disponible sous licence LGPL à l'adresse suivante : `http ://gforge.inria.fr/projects/fdf`. Les composants pour le déploiement d'un grand nombre de technologies sont déjà disponibles. Dans la bibliothèque de personnalités DeployWare on trouve des technologies à base de services (*e.g.* PEtALS JBI, ActiveBPEL, Tuscany SCA), à base de JEE (*e.g.* JOnAS, JBoss, Geronimo), ou de composants (*e.g.* OpenCCM). Des implantations de composants pour l'accès aux machines sont également disponibles : SH, CSH et WinCommand pour le shell, SSH et Telnet pour le protocole, FTP et SCP pour le transfert. La liste exhaustive des composants disponibles dans la bibliothèque de composants DeployWare est disponible à l'adresse suivante : `http ://fdf.gforge.inria.fr`. Une console graphique permet l'administration manuelle des composants une fois

ceux-ci déployés. Elle permet entre autres d'appeler les procédures des différents logiciels, et de modifier les propriétés.

5. Travaux Relatifs

Dans cette section, nous allons détailler quelques travaux actuels traitant, comme DeployWare, la problématique du déploiement automatisé d'applications. Nous positionnons notre approche par rapport à ces différents travaux.

DeployWare permet de décrire avec un haut niveau d'abstraction l'architecture d'un système réparti complexe. Dès lors, il convient de détailler les apports de DeployWare par rapport aux Langages de Description d'Architectures (ADL) (Medvidovic *et al.,* 2000). Il existe une multitude d'ADL différents qui se focalisent sur des préoccupations particulières d'un architecte logiciel. Parmi ceux-ci, on peut citer Darwin (Magee *et al.,* 1996), Rapide (Luckham *et al.,* 1995) ou encore AADL (Feiler, 2003). Certains ADL offrent même des outils formels pour valider les architectures structurellement et comportementalement. Néanmoins, ces langages sont généralement spécifiques à un modèle de composants particulier. En outre, ils se focalisent sur la définition d'architectures de composants logiciels métiers. Enfin, ces langages permettent généralement de décrire la structure de l'architecture, mais pas le protocole de déploiement de cette architecture —*i.e.* comment déployer cette architecture. Ainsi, il n'est pas possible en utilisant ces approches de spécifier l'architecture d'un SRC. Ces systèmes sont par définition hétérogènes en terme de technologies, et concernent aussi bien la couche métier, que les intergiciels et les bibliothèques. DeployWare peut donc être considéré comme un ADL générique et indépendant de la granularité du logiciel à déployer, qui permet de décrire ce que l'on déploie, et comment on le déploie.

Un certain nombre de travaux porte sur la définition d'un modèle de déploiement de systèmes distribués. Notre approche reposant également sur la définition d'un métamodèle, il est donc pertinent de situer DeployWare par rapport à ces approches. Tout d'abord, il existe dans le *Unified Modeling Language* (UML) (Object Management Group, 2006c) un diagramme dit de *déploiement*. Ce diagramme permet de définir des **artefacts** (entités logicielles à déployer), et des **noeuds** (structure d'accueil et d'exécution des artefacts). Des relations de **dépendance** peuvent être établies entre les artefacts. Ce modèle de déploiement se veut générique comme l'est le modèle UML. Néanmoins la généricité dans le modèle de déploiement d'UML est poussée à son paroxysme, à tel point que la sémantique des concepts devient floue. Ainsi la définition d'un modèle de déploiement UML relève plus de la description informelle de l'architecture d'une application répartie, que d'une véritable spécification de déploiement. Dans DeployWare la sémantique des concepts est clairement définie et renforcée par un certain nombre de validations statiques sur les modèles. La description du déploiement d'un SRC avec DeployWare constitue donc une spécification précise du déploiement d'une architecture, prête à être exécutée.

Une autre spécification de méta-modèle a été faite par l'OMG sous le nom de *Deployment and Configuration of Component-based Distributed Applications* (OMG D&C) (Object Management Group, 2006b). Cette spécification a pour ambition de décrire un langage de déploiement et de configuration générique pour applications à base de composants. Un très grand nombre de concepts a été inclus, et il est possible de spécifier la configuration de composants logiciels dans les moindres détails. Néanmoins dans OMG D&C, il est uniquement possible de décrire une architecture de composants logiciels métier, et pas le support d'exécution (intergiciel, bibliothèque). DeployWare permet de décrire le déploiement de toute la pile logicielle d'un SRC.

GADe est un modèle de description générique d'applications à déployer sur des grilles de calcul (Lacour *et al.,* 2005). Le but de ce modèle est de fournir un moyen d'expression générique aux administrateurs d'applications de calcul à grande échelle. Un plan de déploiement générique de l'application peut être généré à partir des descriptions génériques de l'application ainsi que de la grille de calcul, que cette application soit réalisée à base de composants, ou de processus MPI. Dans cette approche un effort a été fourni pour abstraire au maximum la description d'une entité déployable et la rendre indépendante de la technologie. Toute entité est donc modélisée sous forme de processus et de codes à charger. Ce modèle générique reste néanmoins clairement orienté vers les applications parallèles à déployer sur des grilles de calcul. En outre, le déploiement concerne uniquement la couche métier d'une application, et il est impossible en utilisant le modèle GADe de modéliser un SRC complet avec toutes ses couches.

Enfin, un certain nombre d'approches proposent une automatisation du déploiement de SRC, sans reposer sur un méta-modèle clairement défini. Tout d'abord, **Jade** (Bouchenak *et al.,* 2006) est une approche d'administration de systèmes distribués à partir d'une architecture à base de composants. Cette approche est à l'origine du principe de réification d'un système distribué sous forme d'une architecture à composants. Ce principe a notamment été utilisé pour construire des architectures d'administration pour grappes de serveurs J2EE. Néanmoins, aucun effort d'abstraction n'a été fourni pour la description de l'architecture de réification. Ainsi la machine d'administration d'un système d'information doit être assemblée manuellement par le biais d'un langage de description d'architecture à composants dépourvu de sémantique liée au déploiement. Notre approche DeployWare s'inspire également du principe d'architecture réifiant le système, mais offre une surcouche de modélisation spécifique au domaine de déploiement, qui rend en outre possible la validation statique absente dans JADE. Les modèles ainsi réalisés sont projetables automatiquement vers les architectures réifiant le système.

Les outils d'installation de paquets dans certains systèmes GNU/Linux (comme les paquets *.deb* [5] ou *.rpm* [6] selon les distributions) proposent des systèmes de vérification et d'inférence de dépendances. Lorsqu'un nouveau paquet —*i.e.* un nouveau logiciel— est sélectionné comme devant être installé, le système va vérifier à partir de

5. `http ://fr.wikipedia.org/wiki/RPM_Package_Manager`
6. `http ://fr.wikipedia.org/wiki/Deb`

méta-informations fournies pour le paquet si les dépendances sont bien installées sur le système, sinon il propose de les installer également, et récursivement toute l'arborescence de paquets nécéssaires à l'utilisation du paquet selectionné est installée. Certains travaux sur les dépendances entre services, comme **Resolvit** (Ruiz *et al.,* 2004) effectuent des vérifications de dépendances similaires. DeployWare s'inspire également de ce système pour la vérification de dépendances ainsi que l'exécution du déploiement des dépendances. Néanmoins, DeployWare va plus loin dans la vérification en premier lieu parce qu'elle concerne le déploiement réparti dans sa globalité. Le système de paquet GNU/Linux ne concerne qu'un sous-ensemble des tâches de déploiement, en l'occurrence l'installation. En outre DeployWare permet d'autres vérifications que celles liées aux dépendances.

6. Conclusion et perspectives

Dans ce papier, nous avons présenté DeployWare, notre approche à base de modèles pour la description du déploiement de systèmes répartis complexes et hétérogènes. Cette approche repose sur un méta-modèle générique composé de deux préoccupations transverses au déploiement, l'expert logiciel et l'administrateur système. L'expert logiciel décrit le protocole de déploiement d'un logiciel donné. Ce protocole spécifie les propriétés du logiciel, les procédures elle-mêmes composées d'instructions élémentaires (modification de variables d'environnement, lancement d'exécutables, etc.). Ce protocole spécifie également les dépendances vers d'autres logiciels. L'administrateur système décrit le système global, comportant des instances de logiciels tels que décrits par l'expert logiciel. Il configure les propriétés de chacun de ces logiciels ainsi que ses dépendances. Le système ainsi décrit contient le descriptif global de déploiement du système, indépendant de toute technologie. La définition de ce méta-modèle correspond à un (PIM) conformément aux préceptes du MDA de l'OMG. Le caractère dédié du méta-modèle rend possible des validations sémantiques du déploiement. En effet, ce méta-modèle offre des concepts rendant possible la vérification de propriétés essentielles au bon fonctionnement du déploiement décrit. La réversibilité du déploiement, qui assure que tout déploiement est annulable sans effet de bord, la vérification de conformance type/instance de logiciel, la vérification de cohérence des validations, sont autant de valeurs ajoutées à ce méta-modèle générique.

Un modèle de système, une fois validé, peut ensuite être projeté vers une plate-forme d'exécution (*i.e.* un PSM) capable d'exécuter le déploiement du système. Nous proposons également une plate-forme d'exécution pour les modèles DeployWare. Cette plate-forme à base de composants permet d'exécuter le déploiement de systèmes distribués en automatisant la gestion des dépendances et en rendant transparente la gestion de l'hétérogénéité matérielle. En effet, l'expert réseau est en charge de décrire les propriétés des machines dont dispose l'administrateur système. Il configure les propriétés de ces machines et implémente les interfaces d'accès générique à celles-ci (sous forme de composants). Ainsi, de manière transparente les instructions élémentaires des protocoles de déploiement des logiciels lancent des appels sur les

opérations génériques de l'interface, et les commandes sont formattées et envoyées en accord avec les moyens dont dispose la machine. Une syntaxe concrète textuelle permet de décrire le déploiement de logiciels hétérogènes. Les perspectives de ce travail sont nombreuses. Cette plate-forme correspond à un PSM donné, mais nous souhaitons à terme que notre PIM soit projetable vers d'autres plate-formes d'éxécution de déploiement existantes, afin de fournir une architecture conforme au MDA. Dans un premier temps, nous souhaitons adapter notre approche à une problématique sous-jacente du déploiement qui est le déploiement en environnements ouverts distribués. Dans ces environnements, des machines apparaissent et disparaissent dynamiquement, et le déploiement doit donc s'auto-adapter en fonction des fluctuations de l'environnement. Nous souhaitons aborder ce problème en injectant l'expression de politiques d'autonomie (conformément aux principes de l'Autonomic Computing (Kephart *et al.,* 2003)) dans les procédures de déploiement, ainsi que nous l'explicitons dans (Dubus *et al.,* 2007). Cette approche nous permettrait ainsi d'appliquer notre modèle de déploiement aux grilles de calcul, ainsi qu'aux environnements ambiants, qui sont deux exemples d'environnements ouverts distribués (Dubus *et al.,* 2006). Dans un contexte à grande échelle tel que les grilles de calcul, nous souhaitons aller plus loin dans l'analyse statique des modèles de déploiement. Nous souhaitons exploiter un langage de méta-modélisation tel que Kermeta, afin d'ajouter de la sémantique opérationnelle aux concepts de notre méta-modèle. De cette manière, il serait possible statiquement de simuler le déploiement, afin d'analyser le comportement, et d'optimiser son fonctionnement pour passer à l'échelle (modifier les dépendances, paralléliser l'installation de plusieurs logiciels non interdépendants, etc.).

7. Bibliographie

Bieberstein N., Bose S., Fiammante M., Jones K., Shah R., *Service-Oriented Architecture (SOA) Compass : Business Value, Planning, and Enterprise Roadmap*, IBM Press, Octobre, 2005.

Bouchenak S., Palma N. D., Hagimont D., Taton C., « Autonomic Management of Clustered Applications », *IEEE International Conference on Cluster Computing*, IEEE, Barcelona, Spain, Septembre, 2006.

Bruneton E., Coupaye T., Leclercq M., Quéma V., Stefani J.-B., « The FRACTAL Component Model and Its Support in Java », *Software Practice and Experience – Special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, vol. 36, n° 11-12, p. 1257-1284, August, 2006.

Budinsky F., Steinberg D., Merks E., Ellersick R., Grose T. J., *Eclipse Modeling Framework*, Addison-Wesley Professional, Août, 2003.

Dubus J., Merle P., « Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués », *Proceedings of the 1ère Conférence Francophone sur les Architectures Logicielles*, Hermès Sciences, Nantes, France, p. 13-29, Septembre, 2006.

Dubus J., Merle P., « Towards Model-Driven Validation of Autonomic Software Systems in Open Distributed Environments », *Proceedings of the ECOOP Workshop on Model-Driven Adaptation*, Berlin, Germany, Juillet, 2007.

Feiler P., « The SAE AADL Standard : A Basis for Model- Based Architecture-Driven Embedded Systems Engineering », *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, Mai, 2003. Toronto, Canada.

Flissi A., Merle P., « A Generic Deployment Framework for Grid Computing and Distributed Applications », *2nd International OTM Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06)*, vol. 4279 of *LNCS*, Springer-Verlag, Montpellier, France, p. 1402-1411, Novembre, 2006.

Kephart J., Chess D., The Vision of Autonomic Computing, Technical report, IBM Thomas J. Watson, Janvier, 2003. IEEE Computer Society.

Kiczales G., Mezini M., « Separation of Concerns with Procedures, Annotations, Advice and Pointcuts », *in* , A. P. Black (ed.), *ECOOP 2005*, vol. 3586 of *LNCS*, Springer, p. 195-213, 2005.

Lacour S., Christian Pérez T. P., Generic Application Description Model : Toward Automatic Deployment of Applications on Computationnal Grids, Technical Report n° 5733, INRIA, Octobre, 2005. ISSN 0249-6399.

Luckham D. C., Vera J., « An Event-Based Architecture Definition Language », *IEEE Transactions of Software Engineering*, vol. 21, n° 9, p. 717-734, 1995.

Magee J., Kramer J., « Dynamic Structure in Software Architectures », *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of Software Engineering (SIGSOFT 96)*, ACM Press, New York, NY, USA, p. 3-14, 1996.

Medvidovic N., Taylor R. N., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, issue 1, p. 70-93, Janvier, 2000. ISSN : 0098-5589.

Muller P.-A., Fleurey F., Jézéquel J.-M., « Weaving Executability into Object-Oriented Meta-Languages », *Proceedings of MODELS/UML'2005*, p. 264-278, Octobre, 2005. Montego Bay, Jamaica.

Object Management Group, CORBA Components Specification, Available Specification n° formal/2006-04-01, Avril, 2006a.

Object Management Group, Deployment and Configuration of Component-based Distributed Applications, Available Specification n° formal/2006-04-02, Avril, 2006b.

Object Management Group, Unified Modeling Language : Infrastructure, Available Specification, Version 2.0 n° formal/05-07-05, Mars, 2006c.

Poole J. D., « Model-Driven Architecture : Vision, Standards And Emerging Technologies », *ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, Budapest, Hungary, Avril, 2001.

Ruiz J. L., Duenas J. C., Usero F., Diaz C., « Deployment in Dynamic Environments », *DECOR 2004, 1ère Conférence Francophone sur le Déploiement et la (Re) Configuration de Logiciels*, Grenoble, France, Octobre, 2004.

Sun Microsystems Inc., Java 2 Platform Enterprise Edition Specification, Final Release n° 1.4, Novembre, 2003.

Szyperski C., *Component Software : Beyond Object-Oriented Programming*, Addison-Wesley Professional, Décembre, 1997.